4.1 Overview of JavaScript

- Originally developed by Netscape, as LiveScript

- Became a joint venture of Netscape and Sun in 1995, renamed JavaScript

- Now standardized by the European Computer Manufacturers Association as ECMA-262 (also ISO 16262)

- JavaScript can be divided into three categories, core (this chapter), client-side (Chapters 5 & 6), and server-side (not covered in this book)

- We'll call collections of JavaScript code *scripts*, not programs

- JavaScript and Java are only related through syntax

  - JavaScript is dynamically typed

  - JavaScript's support for objects is very different

- JavaScript be embedded in many different things, but its primary use is embedded in HTML documents

4.1 Overview of JavaScript (continued)

- JavaScript can be used to replace some of what is typically done with applets (except graphics)

- JavaScript can be used to replace some of what is done with CGI (but no file operations or networking)

- User interactions through forms are easy

- The Document Object Model makes it possible to support dynamic HTML documents with JavaScript

- *Event-Driven Computation (See Chapter 5)*

  - User interactions with HTML documents in JavaScript use the event-driven model of computation

    - User interactions with form elements can be used to trigger execution of scripts

- *Browsers and HTML/JavaScript Documents*

  - Document head gets function definitions and code associated with widgets

  - Document body gets code that is interpreted once, when found by the browser

4.2 Object Orientation and JavaScript

- JavaScript is NOT an object-oriented programming language

  - Does not support class-based inheritance
    - Cannot support polymorphism

  - Has prototype-based inheritance, which is much different

- *JavaScript Objects*:

- JavaScript objects are collections of *properties*, which are like the members of classes in Java and C++

  - Properties can be data properties or method properties

- JavaScript has primitives for simple types

- All JavaScript objects are accessed through references

- All objects appear as lists of property-value pairs, in which properties can be added or deleted dynamically

4.3 General Syntactic Characteristics

- For us, all JavaScript scripts will be embedded in HTML documents

  - Either directly, as the content of the `<script>` tag whose `language` attribute is set to `"JavaScript"`

```
<script language = "JavaScript">
-- JavaScript script –
</script>
```

  - Or indirectly, as a file specified in the `src` attribute of `<script>`, as in

```
<script language = "JavaScript"
        src = "myScript.js">
</script>
```

- *Identifier form*: begin with a letter or underscore, followed by any number of letters, underscores, and digits

  - Case sensitive

- 25 reserved words, plus future reserved words

- Comments: both `//` and `/* … */`

4.3 General Syntactic Characteristics

(continued)

- Scripts are often hidden from browsers that do not include JavaScript interpreters by putting them in special comments

```
<!--
-- JavaScript script –
//-->
```

(Scripts are not hidden in the examples in the book and in these notes)

- JavaScript statements usually do not need to be terminated by semicolons, but we'll do it

4.4 Primitives, Operations, & Expressions

- All primitive values have one of the five primitive types: Number, String, Boolean, Undefined, or Null

4.4 Primitives, Operations, & Expressions (continued)

- Number, String, and Boolean have wrapper objects (`Number`, `String`, and `Boolean`)

- In the cases of Number and String, primitive values and objects are coerced back and forth so that primitive values can be treated essentially as if they were objects

- Numeric literals – just like Java

- All numeric values are stored in double-precision floating point

- String literals are delimited by either `'` or `"`

  - Can include escape sequences (e.g., `\t`)

  - Embedded variable names are NOT interpolated

  - All String literals are primitive values

4.4 Primitives, Operations, & Expressions (continued)

- Boolean values are `true` and `false`

- The only Null value is `null`

- The only Undefined value is `undefined`

- JavaScript is dynamically typed – any variable can be used for anything (primitive value or reference to any object)

  - The interpreter determines the type of a particular occurrence of a variable

- Variables can be either implicitly or explicitly declared

```
var sum = 0,
    today = "Monday",
    flag = false;
```


4.4 Primitives, Operations, & Expressions (continued)

- Numeric operators - `++`, `--`, `+`, `-`, `*`, `/`, `%`

  - All operations are in double precision
  - Same precedence and associativity as Perl

- The `Math` Object

  - `floor`, `round`, `max`, `min`, trig functions, etc.

- The `Number` Object

  - Some useful properties:
    `MAX_VALUE`, `MIN_VALUE`, `NaN`,
    `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `PI`
    - e.g., `Number.MAX_VALUE`

      - An arithmetic operation that creates overflow returns `NaN`

    - `NaN` is not == to any number, not even itself

    - Test for it with `isNaN(x)`

  - `Number` object has the method, `toString`

4.4 Primitives, Operations, & Expressions (continued)

- *String catenation operator* - `+`

- *Coercions*

  - Catenation coerces numbers to strings

  - Numeric operators (other than +) coerce strings to numbers

    - Conversions from strings to numbers that do not work return `NaN`

- *String properties & methods*:

  - `length` e.g., `var len = str1.length;` (a property)

  - `charAt(`position`)` e.g., `str.charAt(3)`

  - `indexOf(`string`)` e.g., `str.indexOf('B')`

  - `substring(`from, to`)` e.g., `str.substring(1, 3)`

  - `toLowerCase()` e.g., `str.toLowerCase()`

4.4 Primitives, Operations, & Expressions (continued)

- *Conversion functions* (not called through string objects, because they are not methods)

  - `parseInt(string)` and `parseFloat(string)`

    - The string must begin with a digit or sign and have a legal number; otherwise `NaN` is returned

- *The* `typeof` *operator*

  - Returns `"number"`, `"string"`, or `"boolean"` for primitives; returns `"object"` for objects and `null`

- *Assignment statements* – just like C++ and Java

4.5 Screen Output

- The JavaScript model for the HTML document is the `Document` object

- The model for the browser display window is the `Window` object

4.5 Screen Output (continued)

- The `Window` object has two properties, `document` and `window`, which refer to the `Document` and `Window` objects, respectively

- The `Document` object has a method, `write`, which dynamically creates content

  - The parameter is a string, often catenated from parts, some of which are variables

    e.g., `document.write("Answer: " + result + "<br />");`

  - The parameter is sent to the browser, so it can be anything that can appear in an HTML document (`<br />`, but not `\n`)

- The `Window` object has three methods for creating dialog boxes, `alert`, `confirm`, and `prompt`

  - The default object is the current window, so the object need not be included in the call to any of these three

4.5 Screen Output (continued)

1. `alert("Hej! \n");`

   - Parameter is plain text, not HTML

   - Opens a dialog box which displays the parameter string and an `OK` button
     - It waits for the user to press the `OK` button

2. `confirm("Do you want to continue?");`

   - Opens a dialog box and displays the parameter and two buttons, `OK` and `Cancel`

   - Returns a Boolean value, depending on which button was pressed (it waits for one)

3. `prompt("What is your name?", "");`

   - Opens a dialog box and displays its string parameter, along with a text box and two buttons, `OK` and `Cancel`

   - The second parameter is for a default response if the user presses `OK` without typing a response in the text box (waits for `OK`)

→ SHOW `roots.html`

4.6 Control Statements

- Similar to C, Java, and C++

- Compound statements are delimited by braces, but compound statements are not blocks (cannot declare local variables)

- *Control expressions* – three kinds

  1. *Primitive values*

     - If it is a string, it is true unless it is empty or `"0"`

     - If it is a number, it is true unless it is zero

  2. *Relational Expressions*

     - *The usual six*: ==, `!=`, <, >, <=, >=

       - Operands are coerced if necessary

         - If one is a string and one is a number, it attempts to convert the string to a number
         - If one is Boolean and the other is not, the Boolean operand is coerced to a number (1 or 0)

     - *The unusual two*: === and `!==`
       - Same as == and `!=`, except that no coercions are done (operands must be identical)

4.6 Control Statements (continued)

- Comparisons of references to objects are not useful (addresses are compared, not values)

3. *Compound Expressions*

- The usual operators: `&&`, `||`, and `!`

- The primitive values, `true` and `false`, must not be confused with the `Boolean` object properties

  - If a `Boolean` object is used in a conditional expression, it is false only if it is `null` or `undefined`

  - The `Boolean` object has a method, `toString`, to allow them to be printed (`true` or `false`)

- *Selection Statements*

- The usual `if-then-else` (clauses can be either single statements or compound statements)

4.6 Control Statements (continued)

- *Switch*

```
switch (expression) {
   case value_1:
      // value_1 statements
   case value_2:
      // value_2 statements
   …
   [default:
      // default statements]
}
```

- The statements can be either statement sequences or compound statements

- In most situations, the cases end with `break`

- The control expression can be a number, a string, or a Boolean

- Different cases can have values of different types

→ SHOW `borders.html`

4.6 Control Statements (continued)

- *Loop statements*

```
while (control_expression) statement or
                                compound

for (init; control; increment) statement or
                                compound
```

  - init can have declarations, but the scope of such variables is the whole script

```
do
    statement or compound
while (control_expression)
```

4.7 Object Creation and Modification

- Objects can be created with `new`

- The most basic object is one that uses the `Object` constructor, as in

```
var myObject = new Object();
```

  - The new object has no properties - a blank object

  - Properties can be added to an object, any time

4.7 Object Creation and Modification (continued)

```
var myAirplane = new Object();
myAirplane.make = "Cessna";
myAirplane.model = "Centurian";
```

- Objects can be nested, so a property could be itself another object, created with `new`

- Properties can be accessed by dot notation or in array notation, as in

```
var property1 = myAirplane["model"];
```

- If you try to access a property that does not exist, you get `undefined`

- Properties can be deleted with `delete`, as in

```
delete myAirplane.model;
```

- *Another Loop Statement*

  - `for (`identifier in object`)` statement or compound

```
for (var prop in myAirplane)
  document.write(myAirplane[prop] + "<br />");
```


4.8 Arrays

- Objects with some special functionality

- Array elements can be primitive values or references to other objects

- Length is dynamic - the `length` property stores the length

- Array objects can be created in two ways, with `new`, or by assigning an array literal

```
var myList = new Array(24, "bread", true);
var myList2 = [24, "bread", true];
var myList3 = new Array(24);
```

- The length of an array is the highest subscript to which an element has been assigned, plus 1

```
myList[122] = "bitsy";  // length is 123
```

- Because the `length` property is writeable, you can set it to make the array any length you like, as in

```
myList.length = 150;
```

  - This can also shorten the array (if the new length is less than the old length)

  - Only assigned elements take space

4.8 Arrays (continued)

→ SHOW `insert_names.html`

- Array methods:

  - `join` – e.g., `var listStr = list.join(", ");`

  - `reverse`

  - `sort`
    - Coerces elements to strings and puts them in alphabetical order

  - `concat` – e.g., `newList = list.concat(47, 26);`

  - `slice`

```
listPart = list.slice(2, 5);
listPart2 = list.slice(2);
```

  - `toString`
    - Coerce elements to strings, if necessary, and catenate them together, separated by commas (exactly like `join(", ")`)

  - `push`, `pop`, `unshift`, and `shift`

→ SHOW `nested_arrays.html`

4.9 Functions

- `function` function_name([formal_parameters]) {
  -- body –
  }

- Return value is the parameter of `return`
  - If there is no `return`, or if the end of the function is reached, `undefined` is returned
  - If `return` has no parameter, `undefined` is returned

- Functions are objects, so variables that reference them can be treated as other object references (can be passed as parameters, assigned to variables, and be elements of an array)

  - If `fun` is the name of a function,

```
ref_fun = fun;
 /* Now ref_fun is a reference to fun */
ref_fun();  /* A call to fun */
```

- We place all function definitions in the head of the the HTML document, and all calls in the body

- All variables that are either implicitly declared or explicitly declared outside functions are global

- Variables explicitly declared in a function are local

- Functions can be nested, but we won't do it

4.9 Functions (continued)

- Parameters are passed by value, but when a reference variable is passed, the semantics are pass-by-reference

- There is no type checking of parameters, nor is the number of parameters checked (excess actual parameters are ignored, excess formal parameters are set to `undefined`)

- All parameters are sent through a property array, `arguments`, which has the `length` property

→ SHOW `parameters.html` and Figure 4.9

- There is no clean way to send a primitive by reference

  - One dirty way is to put the value in an array and send the array's name

```
function by10(a) {
    a[0] *= 10;
}
...
var listx = new Array(1);
...
listx[0] = x;
by10(listx);
x = listx[0];
```


4.9 Functions (continued)

- To sort something other than strings into alphabetical order, write a function that performs the comparison and send it to the `sort` method

  - The comparison function must return a negative number, zero, or a positive number to indicate whether the order is ok, equal, or not

    - For example, to sort numbers we could define a simple comparison function, `num_order`, as

```
function num_order(a, b) {return a - b;}
```

  - Now, we can sort an array named `num_list` with:

```
num_list.sort(num_order);
```


4.10 An Example

- Function `median`: Given an array of numbers, return the median of the array

```
function median(list) {
  list.sort(function (a, b)
                              {return a - b;});
  var list_len = list.length;

// Use the modulus operator to determine
//  whether the array's length is odd or
//  even
// Use Math.floor to truncate numbers
// Use Math.round to round numbers

  if ((list_len % 2) == 1)
    return list[Math.floor(list_len / 2)];
  else
    return Math.round((list[list_len / 2 - 1]
                     + list[list_len / 2]) / 2);
}  // end of function median
```


4.11 Constructors

- Used to initialize objects, but actually create the properties

```
function plane(newMake, newModel, newYear){
   this.make = newMake;
   this.model = newModel;
   this.year = newYear;
}

myPlane = new plane("Cessna",
                    "Centurnian",
                    "1970");
```

- Can also have method properties

```
function displayPlane() {
  document.write("Make: ", this.make,
                 "<br />");
  document.write("Model: ", this.model,
                 "<br />");
  document.write("Year: ", this.year,
                 "<br />");
}
```

- Now add the following to the constructor:

```
this.display = displayPlane;
```


4.12 Pattern Matching

- Patterns are based on those of Perl

- JavaScript has two approaches to pattern-matching operations, but we will cover just one

- Pattern-matching operations are methods of the `String` object

  1. `search(`pattern`)`

     - Returns the position in the object string of the pattern (position is relative to zero); returns -1 if it fails

```
var str = "Gluckenheimer";
var position = str.search(/n/);
/* position is now 6 */
```

  2. `replace(`pattern`, `string`)`

     - Finds a substring that matches the pattern and replaces it with the string (`g` modifier can be used)

```
var str = "Some rabbits are rabid";
str.replace(/rab/g, "tim");
```

`str` is now `"Some timbits are timid"`
`$1` and `$2` are both set to `"rab"`

4.12 Pattern Matching (continued)

3. `match(pattern)`

   - The most general pattern-matching method

   - Returns an array of results of the pattern-matching operation

     - With the `g` modifier, it returns an array of the substrings that matched

     - Without the `g` modifier, first element of the returned array has the matched substring, the other elements have the values of `$1`, ...

```
var str = "My 3 kings beat your 2 aces";
var matches = str.match(/[ab]/g);
```

       - `matches` is set to `["b", "a", "a"]`

4. `split(parameter)`
   - Like the Perl `split` operator

   - The parameter could be a string or a pattern
     In either case, it is used to split the string into substrings and returns an array of them

     `","` and `/,/` both work

→ SHOW `forms_check.html`

4.13 Debugging JavaScript

- *IE6*

  - Select `Internet Options` from the `Tools` menu

  - Choose the `Advanced` tab

  - Uncheck the `Disable script debugging` box

  - Check the `Display a notification about every script error` box

  - Now, a script error causes a small window to be opened with an explanation of the error

- *NS6*

  - Select `Tasks`, `Tools`, and `JavaScript Console`

    - A small window appears to display script errors

    - Remember to `Clear` the console after using an error message – avoids confusion

